SCIENCE APPLICATIONS Incorporated

(NASA-CR-161335) NASA SOFTWARE SPECIFICATION AND EVALUATION SYSTEM DESIGN Final Report (Science Applications, Inc., Huntsville, Ala.) 20 p HC A02/MF A01 CSCL 09B G3/61

N80-11806

Unclas 46148

SAI-80-743-HU

NASA SOFTWARE SPECIFICATION AND EVALUATION SYSTEM DESIGN

FINAL REPORT

Contract No. NAS8-32526

October 31, 1979

Prepared for:

George C. Marshall Space Flight Center
Data Systems Laboratory
Huntsville, Alabama 35812

Attn: Mr. John Capps, EF15

SCIENCE APPLICATIONS, INC.
2109 W. Clinton Avenue, Suite 800
Huntsville, Alabama 35805
(205) 533-5900

INTRODUCTION

The SSES Managers Guide and the final version of the NSSC-II computer brochure are presented in the first section of this report. Section 2 contains the summary of the work accomplished for each of the scope of work tasks for this contract.

1. SSES MANAGERS GUIDE

SAI has developed all of the following components of the SSES System: a software requirements methodology, a software design language, a structured programming language, a static code analyzer, a dynamic code analyzer, and a testcase generator. Each of these has self explanatory technical and user documentation as listed in Table (1). As far as overall guidelines in managing the personnel of a software project, we refer the interested reader to SAI Programming Guidelines, April/May 1978 SAI Progress Report. This report indicates how to structure the responsibilities of the software implementation and test teams. In terms of overall resources, implementation and test should account for about 50% of the total development effort; the other 50% should go to software requirements and design. For these stages SSES provides a requirements methodology and a design language. The requirements approach is a modified HIPO-type description of the requirement functions and subfunctions. In order to identify these functions, we recommend the creation of Data Flow Diagrams (c.f. DeMarco, Structured Analysis and Design, Yourdon Press) which use data as a starting point. Once all the data flows have been established, the processing points can be elaborated using the SSES requirements methodology. Another advantage of using DFD's is that they lead to a natural high level software design - as illustrated in Figure (1). Once established, this design can be expressed in terms of SSL - the SSES design language.

One last point we should make is about the cost and scheduling of software. The very best reference we can give on that topic is an article by Alvin L. Kustanowitz, which we include in the appendix of this report.

In retrospect, we think there are a wide variety of reasons why an integrated system of software development tools should be employed - for increased reliability, quicker development, better maintainability, etc., but one reason which should stand out to all is lower cost. Attesting to this fact is Table (2) which gives productivity figures obtained while developing the various SSES tools - along with a pilot project - a relational data base management system. These programs were developed using some or all of the SSES tools. Once the SSES developed programs are categorized as to type of development, their productivity figures compare favorably with industry standards. Even the relational DBMS program - which was a highly theoretical, non-standard development - had favorable results. We believe that through using the tools of the SSES system, predicting productivity of 10 lines/day for any HOL program will be an entirely safe and certain estimate.

TABLE 1

TECHNICAL DOCUMENTATION FOR SSES COMPONENTS

SSES Component	Design Document	User's Manual	Operation Guide	Listing	Flowcharts
Software Requirements Methodology	Software Requirements Methodology Design Specifications SAI-78-581-HU				
Software Specification Language	NASA Software Specification Language Translator Unit Module Descriptions SAI-78-594-HU	Introduction to Formal Specification Technique and SSL SAI-78-504-HU	NASA Software Specification Language Operation Guide SAI-78-597-HU	Separate documentation with no title	NASA Software Specification Language Translator Flowcharts SAI-78-595-HU
Structured FORTRAN Preprocessor	NASA Structured FORTRAN Preprocessor Unit Module Descriptions 1. SAI-78-601-HU	NASA Structured FORTRAN Preprocessor User's Manual SAI-78-600-HU	Included in User's Manual	Separate documentation with no title	NASA Structured FORTRAN Preprocessor Flowcharts SAI-78-602-HU
Static Analyzer	FACES Unit Module Descriptions SAI-78-584-HU and Updates to Existing FACES Documentation SAI-78-586-HU	Updates to Existing FACES Documentation SAI-78-586-HU	No changes to existing documentation	Separate documentation with no title	FACES Flowcharts SAI-78-585-HU
Data Base Verifier	Data Base Verifier Design SAI-78-583-HU				

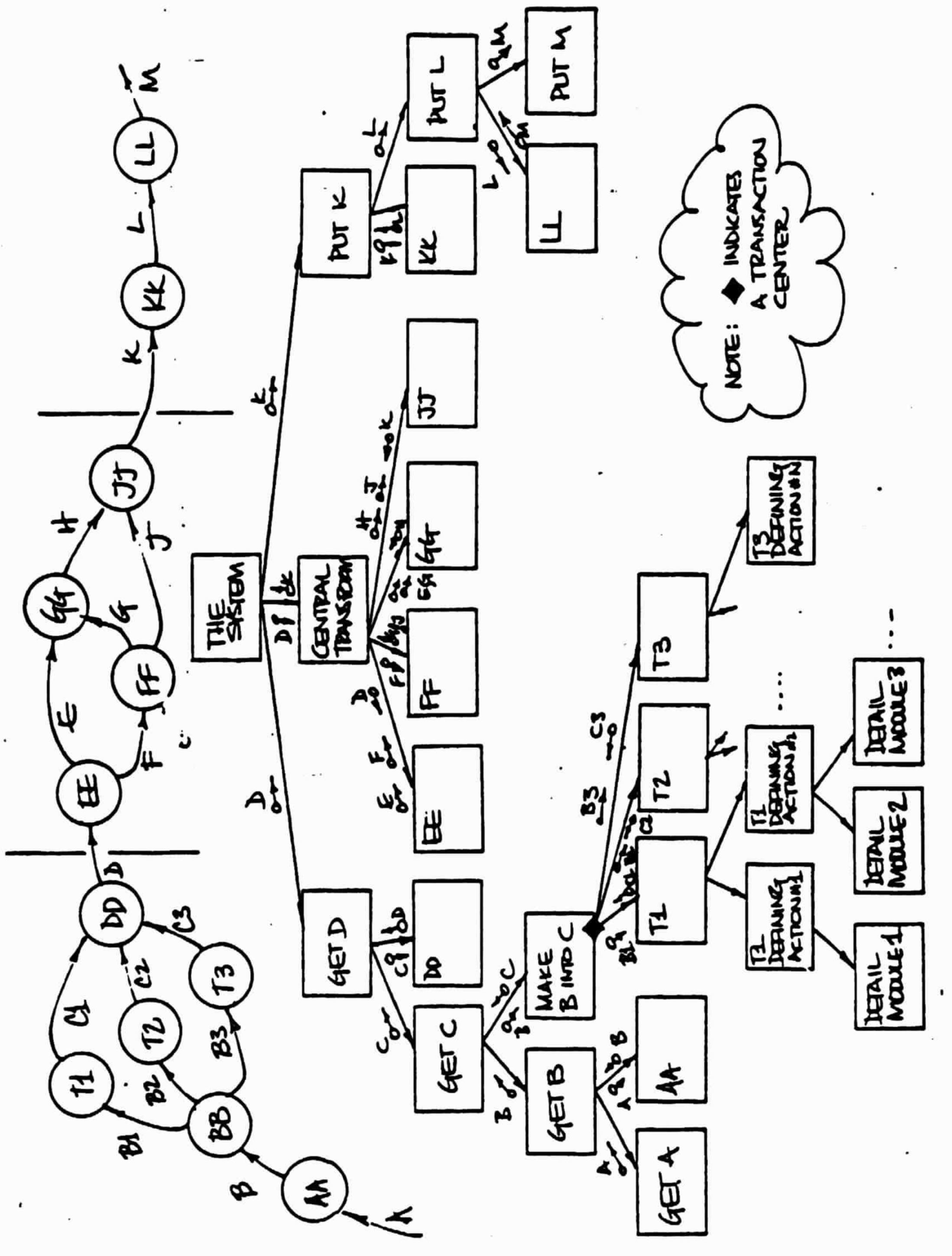
TABLE 1 (Cont'd)

Dynamic Analyzer	NASA Dynamic Analyzer Detailed Design Document Version II Revision 0 SAI-78-578-HU and Dynamic Analyzer FORTRAN Data Base SAI-78-580-HU	NASA Dynamic Analyzer and Structural Analyzer User's Manual SAI-78-577-HU	Included in User's Man-ual	Separate docu-mentation with no title	Included in an Appendix to Design Document SAI-78-582-HU
Structural Test Case Generator	NASA Structural Analyzer Extension to Dynamic Analyzer Detailed Design Document SAI-78-579-HU	NASA Dynamic Analyzer and Structural Analyzer User's Manual SAI-78-577-HU	Included in User's Manual	Separate docu-mentation with no title	Included in Design Document

1. Other design documentation includes:

   Calling Hierarchy for Modules Constituting the NASA Structured FORTRAN Preprocessor
   COMMON Names and COMMON Variables Referenced in the NASA Structured FORTRAN Preprocessor
   Cross Reference of Modules and COMMON Names in the NASA Structured FORTRAN Preprocessor

   SAI-78-603-HU
   SAI-78-604-HU
   SAI-78-605-HU



AA
BB
T1
T2
T3
DD
EE
FF
GG
JJ
KK
LL
THE SYSTEM
GET D
CENTRAL TRANSFORM
PUT K
GET C
DD
EE
FF
GG
JJ
KK
PUT L
LL
PUT M
GET B
MAKE B INTO C
GET A
AA
T1
T2
T3
T1 DEFINING ACTION #1
T1 DEFINING ACTION #2
T3 DEFINING ACTION #N
DETAIL MODULE 1
DETAIL MODULE 2
DETAIL MODULE 3
NOTE: ◆ INDICATES A TRANSACTION CENTER

Figure 1

TABLE 2. CODE LINES/DAY OVER ENTIRE DEVELOPMENT

	Aron(1) (HOL) (No System Test)	Aron(2) (Assembly Language)	Corbato(1) (Assembly Language)	SSES(3) (HOL)
Very Few Programmers or Program Intersections	39	20		50(SSL)
Some	19	10		12(Relational DBMS)
Many	6	5	5	17(Dynamic Analyzer)

(1)Brooks, "Mythical Manmonth"

(2)Aron, "Estimating System Costs"

(3)SSES Project Data

The *NSSC/II* is the only computer that is space-qualified. It is lightweight –– less than 30 pounds –– and has lower power requirements –– using 28 volts.

The *NSSC/II* has been designed to be extremely tolerant of ambient conditions, meeting temperature extremes with no performance degradation.

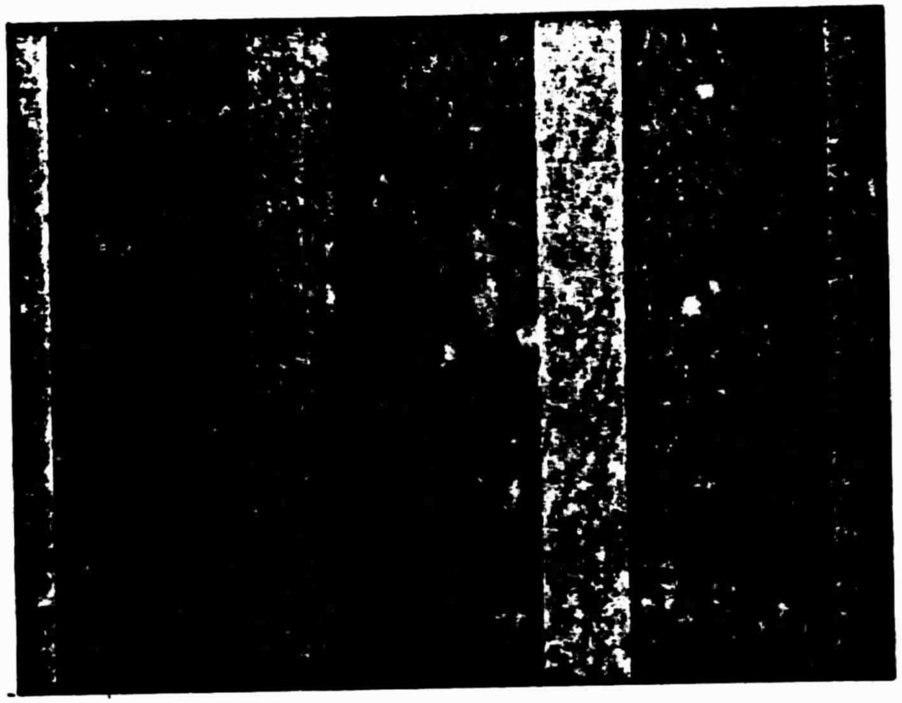
The *NSSC/II* interfaces to peripherals in a wide variety of ways including direct memory access.

The *NSSC/II* is easy to program, supporting two high level languages, FORTRAN and HAL/S, as well as an IBM 360 compatible assembly language.

NSSC/II

NASA Marshall Space Flight Center
Data Systems Lab. Bldg. 4708
Huntsville, Alabama

NSSC/II. . .



NASA STANDARD SPACECRAFT COMPUTER

- Space Qualified/SPACELAB Integrated
- Lightweight
- Low Power
- IBM 360 Instruction Set
- High Order Language Support

Externally Attractive

PHYSICAL CHARACTERISTICS

• Length	—	38.6 cm
• Width	—	39.8 cm
• Height	—	13.919 cm
• Volume	—	11,152 cm^3
• Weight	—	13.2Kg

POWER — 28 ± 7 Vdc, 155W for 32K bytes

PERFORMANCE

• Short Word	—	350K ops/sec
• Full Word	—	200K ops/sec
• Floating Point	—	40K ops/sec

MEMORY OPTION

- 32K byte basic configuration, expandable to 1 megabyte
- Fault Tolerant Memory
- Power Switched Core

I/O

- Direct, buffered, DMA for either System and Test Support Equipment

INTERRUPTS

- I/O
- Program
- Supervisor
- External
- Machine Check

TIMER

- Real Time
- 16 Bit Interval

Convenient & Plentiful Support Software

OPERATING SYSTEM

- Multitasking, supports FORTRAN, HAL–S

ASSEMBLER, LINK EDITOR

- IBM 360 Compatible

PROGRAMMING LANGUAGES

- FORTRAN G/H
- HAL–S

SIMULATOR

- *NSSC/II* Simulation on the 360

PAPER/MAG TAPE FORMATTER

- *NSSC/II* Execution of 360-generated Load Module

UTILITIES

- IBM/SAI Functions and Macros
- *NSSC/II* 360 Dumps, Snaps, Traces, Display Registers, Searches, and Restore Memory

Rugged Environment Characteristics

RELIABILITY

- 12,400 Hr. MTBF (64K byte)

PARTS

- MIL–STD–975 Grade 2

ENVIRONMENTAL ACCEPTANCE LEVEL

- -20^oC to $+60^oC$, Cold Plate
- Random Vibration to $.04G^2$ Hz

2. QUANTITATIVE PROGRESS

SOW Task	% Completed	Action Taken
A. Develop Pilot Software	100	We have built a relational Data Base Management System using SSES.
B. Test and Evaluate SSES Components	100	An SSL report, written by an independent evaluator, was submitted in our July 1, 1977 Progress Report. Also, the dynamic analyzer, static analyzer and structured preprocessor have been tested and evaluated at another NASA computer center at MSFC as well as the John Hopkins Applied Physics Lab.
C. Modification of SSES Components	100	FACES, the structured FORTRAN preprocessor, and two versions of the dynamic analyzer have been converted to the UNIVAC 1108. SSL was modified to improve error recovery. Extending the analyzing capabilities and decreasing the execution overhead was accomplished for the dynamic analyzer.
D. Advanced Research	100	Documented in our August 1977 Progress Report was an investigation of the SSES system towards microprocessor software. Our extensive research into the data bases was partially reflected in our October (1977) Progress Report. A paper entitled, "Computer Program Development Analysis" was written which appears in Appendix A of the September 1978 report.
E. NSSC-II Software Documentation and HAL/S Software/ Assessment	100	Documentation has been completed.
F. HAL/S and FORTRAN Comparison	100	Small comparison program was run.
G. NSSC-II Documentation	100	Documentation is completed and has been printed.

APPENDIX

SYSTEM LIFE CYCLE ESTIMATION (SLICE): A NEW APPROACH TO ESTIMATING RESOURCES FOR APPLICATION PROGRAM DEVELOPMENT

SYSTEM LIFE CYCLE ESTIMATION (SLICE): A NEW APPROACH TO ESTIMATING RESOURCES FOR APPLICATION PROGRAM DEVELOPMENT

Alvin L. Kustanowitz

IBM Corporation
Data Processing Division
White Plains, New York 10604

ABSTRACT

This paper presents a technique for the accurate estimation of manpower required to implement programming applications, from simple batch programs to complex, on-line systems in both the conventional and top-down, structured programming environments.

The history of estimating techniques is reviewed in order to show why most of them have failed to produce accurate and consistent results, and to demonstrate the need for a fresh approach to resource estimation.

The technique, System Life Cycle Estimation (SLICE), has greatest validity when a system design already exists, but may be used, although with less accuracy, at earlier stages of development.

THE HISTORY OF ESTIMATING TECHNIQUES

Ever since the appearance of the first programmable computer, we have been trying to predict, forecast and estimate how long it will take and how much it will cost to develop programs and programming systems. Almost without exception, these efforts have failed. Certainly, some well-conceived estimating techniques have been published because their developers found them to be useful in a particular environment. In most of these cases, the authors were very careful to describe the environment for which the technique was suited, and to advise potential users of its limitations and restrictions.

J. D. Aron, in a paper that has become a classic in the field[1], suggests a quantitative approach using 20 assembly-language source statements per day for "easy" programs, 10 per day for "medium" programs and 5 per day for "hard" programs. While these figures have been widely used and quoted, applying them to a typical program or system doesn't always result in accurate estimates. Careful reading of the paper shows why--the figures are applicable only for "large" systems, where these are defined as requiring:

- More than 25 programmers
- More than 30,000 deliverable instructions
- More than six months development time
- More than one level of management

Furthermore, the definitions of "easy", "medium" and "hard" are themselves revealing.

EASY	- Very few interactions with other systems elements. The class includes most problem programs or "application" programs. Any program the main function of which is to solve a mathematical or logical problem is probably in this class. Easy programs generally interact only with input/output programs, data management programs, and monitor programs.
MEDIUM	- Some interactions with other system elements. In this category are most utilities, language compilers, schedulers, input/output packages and data management packages. These programs interact with hardware functions, with problem programs, with monitors, and with others in this class. They are further complicated by being generalized enough to handle multiple situations; e.g., I/O from many different I/O devices or management of data files with variable numbers of indices.
HARD	- Many interactions with other system elements. All monitors and operating systems fall in this class because they interact with everything. Special purpose programs, such as a conversational message processor, may be in this class if they modify the master operating system.

It becomes apparent that most applications which a company is likely to undertake, such as payroll, personnel, inventory, sales analysis, accounts receivable and payable and even on-line systems developed with the aid of teleprocessing control systems and data base management systems, fall into the "easy" category. Only if the effort is one of developing a package like a data base management system does the degree of difficulty move up into the "medium" category.

Clearly, while this approach may have been useful in its context, which is the development of such systems as the SABRE airline reservation system and the Operating System (OS) for IBM's family of 360 and 370 computers, it is not directly applicable to the wide range of programs which a typical company is likely to design and implement.

In "Management Planning Guide for a Manual of Data Processing Standards"[2], there is a section called "Technique for Estimating Project Duration." In it, the user is led through an elaborate scheme of applying weighting points for program complexity, input/output characteristics, major processing functions, programming know-how and programmer job knowledge.

Program development time is computed by multiplying the sum of the first three sets of weights by the sum of the last two sets. After calculating this to two decimal places, we are told to add an additional 70 to 110% for "other system time."

The use of decimal places in these calculations is particularly subject to misinterpretation because precision usually implies accuracy, which is clearly not the case here.

If these published techniques don't work for typical business applications, how then, do we estimate the resources required?

SYSTEM LIFE CYCLE ESTIMATION (SLICE)

The Basic Method

All programming projects have one thing in common: A Life Cycle. They begin, and sooner or later, they end. Between the starting and ending points, the development effort proceeds through a succession of distinct phases. The number and composition of these phases will vary, depending on the size and complexity of the project, and the installation standards and procedures for such activities as project management, program and system testing and documentation.

Most attempts to produce "standard" estimating techniques have failed because of the impossibility of imposing one organization's standards on others. Another point of difference is to what part of the development cycle we are applying an estimating technique.

The point here is that, unless all those trying to estimate the development cost of a project agree on the system life cycle or profile for this project and to which portion of the cycle they are applying productivity factors, the only result is confusion.

Of course, it must be understood that any estimates will be virtually useless unless an organization has implemented and is committed to project control standards. Without such standards, changes are likely to occur during the development cycle which will distort or even totally invalidate the original estimates.

Ask any programmer of programming manager to describe the stages of development of a programming system. The most likely answers will be:

```
DESIGN--CODE--TEST     DESIGN--CODE--TEST
                   OR
 30%----40%----30%      1/3----1/3----1/3
```

These classic distributions, although universally used, are actually meaningless when applied to one particular project. If the project consists of a single program to do a simple summation of numbers and print out the result, the distribution is more likely to be 10-80-10 than 30-40-30 because design and testing are trivial while most of the work will be in coding and compiling. At the other extreme, if the project is an on-line real-time airlines reservation system or all the programming required for a manned landing on Mars, the distribution might be 45-10-45, with the coding a relatively small part of the total effort compared to the years of design and extensive multiple levels of unit, integration, system and regression testing.

This continuous range of distributions for projects of different size is shown graphically in Figure 1.

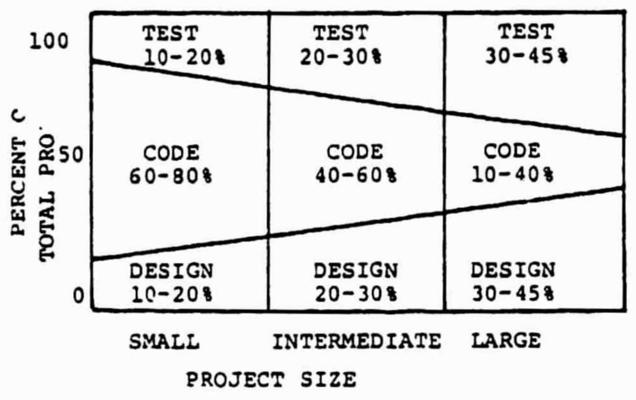


FIGURE 1. SYSTEM LIFE CYCLE VARIES WITH PROJECT SIZE

The essence of System Life Cycle Estimation is the realization that no two system development efforts are the same, especially when they are implemented in different organizations; however, if multiple systems are developed in the same environment, they will share many characteristics and the experience gained in the first, if quantified through accurate record-keeping techniques, will serve as a sound basis for estimation of its successors.

Once the estimator accepts this line of reasoning, he can proceed to define a realistic model of the proposed system in its real environment, apply known percentages and productivity factors and translate the raw number of technical man-days so derived into a time-phased project plan.

How To Use Slice: A Step-by-Step Approach

Describe Your Project Life Cycle

In the preceding section, we discussed the relative proportions of design, code and test in different projects. Usually, these three phases are further broken into smaller components such as:

Planning
Feasibility Study
Requirements Definition
Conceptual Design
Program Design
Data Base Design
Program Specifications
Program Flowcharting
Coding
Compilation
Data Base Creation
File Conversion
Unit Test
Integration Test
System Test
Documentation

The first setp in using System Life Cycle Estimation is to construct, from the categories listed above, and any others you may add, a project profile describing system development as you see it based on actual operation of your company, division, group or department, and based on previous projects completed in the same environment. Project size should have little bearing on this step. Here are some examples:

PROFILE A

1. Functional Requirements Study
2. Conceptual System Design
3. General System Design
4. Program Specifications
5. Coding
6. Compilation and Unit Test
7. System Test

PROFILE B

1. Planning
2. Program Design

3. Data Base/File Design
4. Programming
5. Data Base Creation
6. Testing

Would you select either of these project profiles as your own? Probably not exactly --you'll want to make some adjustments, but you'll end up with between 6 and 10 distinct phases of a system development cycle--and the best part of it is that you are not being forced to accept anyone else's idea of what phases make up a total project plan--this is your plan for your project in your company.

Assign Percentages to Each of the Phases of Your System Life Cycle

This is a bit harder to do. It all depends on how accurately you have kept records on previous projects. Also, unless you have a very specialized group, you are likely to have more than one type of system life cycle, e.g. small batch systems, intermediate batch systems, large batch systems, small on-line sys-ems, intermediate on-line systems, large on-line systems, etc.

What do you do if you haven't ac-cumulated enough historical data to assign percentages to each phase? You start right now and collect as much data as you can for future estimates. In the mean-time, you have to work with what you have. This may well be only rough "guesstimates" --maybe not the most accurate information, but if you apply these to a profile in which you have confidence, you're already far ahead of the old way of pulling numbers out of the air.

Remember, once the model is built, it's easy enough to change and refine the percentages as you learn more about your project life cycle.

At this point, your life cycle profile might look something like Figure 2 for a small batch system or Figure 3 for a large on-line system.

START	FUN	CON	DES	SPC	COD	UNI	SYS	END
	.11	.05	.11	.23	.11	.23	.16	

FIGURE 2. Typical Small Batch System Life Cycle

START	FUN	CON	DES	SPC	COD	UNI	SYS	END
	.18	.09	.18	.10	.06	.09	.30	

FIGURE 3. Typical Large On-Line System Life Cycle

where FUN = Functional Requirements Definition
CON = Conceptual System Design
DES = System Design
SPC = Program Specifications
COD = Coding
UNI = Unit Test
SYS = System Test

Select Productivity Factors

How many instructions per day can you expect your programmers to produce? You will probably have more than one number here--the programming language used will be a factor. The key point here is: Only you know your environment. Since you have probably implemented projects before, all that needs to be done is to add the lines of code (source or delivered instructions--it doesn't matter which you use--as long as you are consistent) and divide by the number of man-days reported on the project.

The strongest objections are likely to be raised here in the form of:

- "But what if I don't have data from previous projects?"

The only answer I can give is: You should have kept detailed records, but if you haven't there is no better time than now to begin.

- "Why instructions per day? How can I expect my programmers to turn out a fixed number of lines of code every day? How can they sustain such a daily rate of production?"

This is not an easy question to answer. Until recently, no better unit of measure-ment has been found.

Lately, there has been increasing discussion of an alternate measure, person-months (or days) per unit (e.g. 1000 lines) of code. This measure seems to have great merit, especially when applied to large systems, particularly where there is a high degree of scaffolding and much non-coding activity must be factored into overall costs. The typical computer user, however, tends to feel more comfortable with the traditional lines per day and this is still a very useful measure at the low and intermediate part of the scale.

For these applications, where the great majority of project personnel are analysts and programmers, it is still the easiest to relate to.

After all, when a system is delivered, the source code is there for all to see and it is preserved as part of the documentation. The only other data that has to be kept is an accurate report of hours or days spent in programming development. This data is vital if you ever want to control your projects rather than have your projects control you.

Establish Estimating Basis

Let's say you have concluded that reasonable productivity factors for your installation are 18 lines of code per day for COBOL (which may generate from 40 to 60 actual deliverable instructions) and 25 lines per day for Assembler. These numbers are meaningless without one more piece of information--over what part of your system life cycle do these factors apply?

Take a 1000 line program as an example. If you mean 18 lines per day for the entire project from start to finish, the estimate is 1000/18 or 56 man-days. However, if you are measuring 18 lines per day from the start of the program specifications through the end of unit test, for the large on-line system profiled in Figure 3, the 56 man-days accounts for only the middle portion, or 25% of the system life cycle. The total technical man-days would be 56/.25, or 224.

Again, let me emphasize that you can estimate either way, as long as you apply the factors consistently.

Estimate the Total Number of Instructions in the Finished System

This is not as difficult as it may seem. Of course, if you don't have the slightest idea of where to begin, you shouldn't be estimating yet. A good deal of design work needs to be done. The earliest point at which an estimated instruction count can be made is at the completion of a conceptual design. A re-estimate should be made at the end of detailed program design and after program specifications and/or flowcharts are done. You will then be able to refine the accuracy of your earlier project estimates.

Any experienced programmer should be able to guess the approximate number of lines of code from a combination of his previous experience, knowledge of other programs of various sizes, and a look at the specifications or flowchart or narrative of a proposed new program. It may not be a guarantee of actual final program size, but it's the best, most consistently accurate, self-correcting measurement criteria available.

Calculate Technical Man-Days

Divide the total number of estimated instructions by the productivity factor, e.g.

$$\frac{\text{1000 instructions}}{\text{18 instr/day}} = \text{56 man-days, or person-days, if you are so inclined}$$

If the 18 instructions/day factor was developed to apply over your entire system life cycle, stop here. You have the total technical man-days. If the instructions per day factor applies only to a percentage of your total system life cycle, divide the man-day figure by the percentage. For example, if it applies over 50% of the cycle, divide 56 by .50 to get 112 man-days; if it applies to 63% of the cycle, divide 56 by .63 to get 89 man-days.

This technique works well in a steadystate environment, where the project being estimated represents no great changes in approach or technology such as the first data base system or the first use of structured programming in the organization.

Caution must be taken when the development environment is in a state of transition. If, for example, we have found a way to double the productivity of coding and unit test without affecting the duration of the other project phases, we must reassess the life cycle profile. It must be modified to reflect a lower percentage for the phases in which productivity has increased and a higher percentage for the others. Otherwise, the casual application of ratios could inadvertently reduce the time allotted for all phases, not just those which benefited from a productivity increase.

Translate into Time-Phased Project Plan

Still with me? Good! We're almost finished. The last step is to go from a raw figure of technical man-days to a time-phased project plan. Users of this technique have found that it works best for a "square-root" manpower vs. time distribution. For example, if your total technical man-days comes out to be 720, which divided by 20 days per month yields 36 man-months, the project should take 6 people 6 months to complete. If you try to complete the project in one month using 36 people or in 36 months with one person, you're not likely to make it within the 36 man-month estimate.

The total effort expended will inevitably be somewhat greater, in the first case because of the increased interaction between 36 people and the sharing of other limited resources such as computer time. In the second case, three years is a long time and you can be sure that design changes and personnel changes will lengthen the project.

After a while, you'll be able to adjust the basic technical man-days based on these types of time frame or manpower constraints without too much trouble. Other "extras" you may want to consider in arriving at a realistic total cost for the project are project management and documentation, if these are not already part of your system profile. How many man-days you allocate for these functions is highly subjective and varies considerably from one installation to another.

The Impact of Data Base/Data Communications Program Products and Interactive Program Development

The use of these system development tools can easily be taken into account with System Life Cycle Estimation. They are likely to impact project development in two ways:

- Reducing the number of lines of code required to be written, because Data Base/Data Communications systems provide many of the data management functions which would otherwise have to be designed and programmed each time.

- Increasing the number of lines of code per day which a programmer can be expected to produce because of the reduced testing and debugging turnaround time provided by interactive development.

With System Life Cycle Estimation, just know which of these tools you will be using and take them into account in setting up your productivity factors and in estimating the total lines of code to be produced.

Estimating in a Structured, Top-Down Environment

If your project is being developed using one or more of the Improved Programming Technologies (Structured Programming, Top-Down Development, HIPO, Chief Programmer Team Operations, Development Support Libraries and Structured Walk-Throughs) System Life Cycle Estimation is just as valid as it is in the conventional environment. The top-down approach usually alters the shape and composition of a typical project profile. The major differences are:

- Detailed design, coding, unit test, integration test and documentation tend to overlap and should be considered as a single phase in system development. Therefore, these types of projects will have fewer development phases, e.g.

  1. Feasibility Study
  2. System Design
  3. Top-Down Development
  4. System Test

- The percentage of total effort devoted to system test will be reduced, because the top-down approach brings a more fully checked-out system into the system test phase.

- The productivity factors (instructions per day) are likely to be substantially higher.

- The System Design phase is typically longer in this environment.

CONCLUSION

Now you're an expert in System Life Cycle Estimation. It's easy to use, and it works. You can do the calculations manually, or if you're skilled in programming in any interactive language (e.g. APL or BASIC), you can write a simple program to accept input values for project profile, language type, number of instructions and productivity factors, and print out the estimates automatically.

The only condition of use which I ask you to accept is: If you are not satisfied with SLICE's predictive accuracy the first time you use it, don't reject the technique yet--try it a second time.

After you have properly applied the technique at least twice, I estimate that it will become a permanent part of your own life cycle.

REFERENCES

1. Aron, J.D. In Estimating Systems Costs, Fourth Annual Regional Seminar of Tennessee State Chapter of Association for Systems Management, February 13, 1971.

2. Management Planning Guide for a Manual of Data Processing Standards, Form C20-1670-2. White Plains, New York. IBM Corporation, February 1971.